DynJava: Type Safe Dynamic Code Generation in Java

Yutaka Oiwa Hidehiko Masuhara Akinori Yonezawa

University of Tokyo

Abstract

Dynamic code generation is a technique that generates and executes fragments of executable code during the run-time of an program to reduce the execution time of programs. However, most of currently available dynamic code generation systems have weakness on high-level language support for describing dynamic code fragment. Especially, safety of dynamic composition of code fragments is not sufficiently supported. To solve this, we present a Java-based strongly-typed language for dynamic code generation. This language gives precise types to dynamic code fragments, to guarantee the type-safety of dynamically-composed codes.

1 Introduction

Dynamic code generation (DCG) is a technique that generates and executes fragments of executable code during the run-time of a program [11, 9]. It is useful for reducing execution time of programs, because generated fragments can be optimized by using run-time values, which are not available at static compilation-time. Since generated code holds runtime values such as number of an iteration or a condition of a branch as constants, it may run faster than statically-generated code.

There is three ways to implement DCG. The first is to directly write down a program which generates native machine instructions on a memory [5, 4]. However, writing a correct program in this approach is extremely difficult and error prone. The second is to automatically generate a program that generates an optimized version of a given generic program—so called *(run-time) partial evaluation* [9, 2]. As this approach relies on static analysis to determine where to optimize, resulted optimization would be too conservative. The third is to define a language in which the programmer can write a fragment of dynamic code as an expression in a high-level language. This approach can solve the problems of the former two approaches. Our *DynJava* is based on this approach.

DynJava is an extended language to Java in which the user can write dynamic code fragments in the syntax that is similar to the Java's. In addition, dynamic code fragments in DynJava are statically typed, and type safety of dynamically composed code is statically guaranteed. We devised a type system for DynJava, and implemented a system that translates DynJava program to a Java program.

2 The DynJava Language

The DynJava language has, in order to generate new classes, a few constructs for combining several parts of dynamic code fragments together. This section presents a brief overview of DynJava.

2.1 Code fragments and contexts

In DynJava, the user can declare a *code specifications* to define a dynamic code fragment, whose syntax is similar to `cspec` in ‘C [11]. They are combined to generate an anonymous subclass of an abstract class.

To check the type-safety of the code generated by dynamically code compositions described later, the code specification's type includes its context information.

A context information of a code specification is very precise in DynJava. It includes following information:

Name of the base class (superclass) Because all Java program codes are enclosed in methods of some classes, there is always a "current class" and its superclass for every statement and every expression. In an instance method, names of the methods and fields in the current class are visible, and `this` can be used as an read-only value of that class type. We call this "context (or environment) is an *instance context*." In the class method, only class methods and class fields are accessible through simple name, and `this` is inaccessible. We call this "*class context*".

Because current class of dynamically-generated code is anonymous in DynJava, a reference to an instance of dynamically-generated class is always used as a value of its superclass's type, whose definition is statically available. We name this "base class". A context holds the name of the superclass, with a flag whether the context is instance context or class context.

The base class of an instance context is notated as "`extends` *classname*", and those of an class context as "`static extends` *classname*".

Variables and fields of current class A *variable context* holds types of all variables which is visible in the current scope. In this type system, a variable context also holds all fields which are accessible by simple names, and by `this.`*name* notation.

The entry of the simple name in the variable context is notated as "*type name*" (ex. "`int i`" or "`Vector v`"). The entry of the field of `this` is notated as "*type* `this.`*name*".

Note that if some name is available through `this`, the name is always accessible through its simple name. However, type of the *name* and `this.`*name* may be different, as local variable may hide instance fields and class fields.

If a context has a base class, the type-checker automatically consults the definition of the base class and expands its fields to the variable context.

Methods and Constructors A *method context* holds the list of methods and constructors which are defined in or inherited by the current class. For each method, a method context holds 1) the name of the method, 2) type of the return value, 3) types of the formal parameters, and 4) list of exception types which may be thrown by the method. For each constructor, context holds 1′) a flag whether the constructor is defined in current class or in direct superclass, and 3) and 4).

Method entries in a method context are notated as "$type_r$ *name*`(`$type_1$`,` $type_2$`,` …`) throws` exn_1`,` exn_2`,` …", where $type_r$ is the return type, $type_1$, $type_2$, … are the types of the formal parameters, and exn_1, exn_2, … are the types of the exceptions which may be thrown. For example, `clone` method defined in the `Object` class in the JDK is notated as "`Object clone() throws CloneNotSupportedException`". For constructor entries, return types are omitted and either `super` or `this` are used in place of *name*.

As same as variable context entry, entry of methods and constructors which are defined in the superclass are expanded automatically by the type checker.

Type of return value The type of the return value depends on the type of surrounding method. This information is notated as "`return` *t*". The context in which only `return` statements without argument can be used is notated "`return void`".

Context is allowed to contain no return type specification. Code specification with such context can not contain any "`return`" statement in it, but can be used in places whatever type of return value is required.

`break`-able Labels A *label context* holds all labels which is available for use with `break` or `continue` statement. `continue`-able label is provided by `while` and `for` statements, and `break`-able label is provided by those and `switch` statements. In addition to this, those statements without label is treated to be providing null-label. `continue`-able label with tag *t* is notated as `continue` *t*, and `break`-able label of those as `break` *t*. Null-labels are notated simply as `continue` and `break`.

Throwable exceptions At each point in the programs, *exception context* holds a set of exception types which are handled by either method caller or try-catch clause. For example, at the point of star (⋆) in the following program fragment, exceptions of any subclass of the class `java.io.IOException`, `ClassNotFoundException` can be thrown.

```
void method_1(int x) throws IOException {
    try { ⋆ }
    catch (ClassNotFoundException e) { ... }
}
```

In a program text, this information is notated as "`throws` *e*", where *e* is the name of the exception class.

Other information of context There are two miscellaneous states which a context holds. First, "constructor state" means that the current block is used as a body of a constructor, and that some constructor of the current class or of direct superclass must be explicitly invoked at the top of current block. This state is notated as "`super`". Second, "switch state", notated as "`switch`", means that the current block is used as a body of a `switch` statement, and the special labels `case` *i*`:` and `default` can be used.

For example, the context specification <`int x`> specifies contexts where `x` is bound to a `int` value, and <`return int`> specifies that contexts are methods that return `int` values.

2.2 Code specifications

DynJava has two kinds of code specifications, *statement specifications* and *expression specifications*. Each of them corresponds to a Java's statement or expression, respectively. The type of a statement specification is written as `code_spec`<Γ>, where Γ is the context on which the body of the specification depends. The type of an expression specification is written as *t* `exp_spec`<Γ>, where *t* is the type of values generated by evaluating a generated code fragment from the specification. The type *t* is called a target type of the expression statement type.

A code specification begins with a backquote (`‘`), followed by a context specification and either a statement or an expression. Statement specifications have the form "`‘`<*context*>`{`*body*`}`", and expression specifications have the form "`‘`*type*<*context*>`(`*body*`)`".

Code specifications can have free (unbound) variables, if they are declared in its context specifications. The following is the examples for code specifications.

```
‘<String x>{ System.out.println("hello, " + x + "!"); }
‘double<int x>(x + 1.5)
```

When a code specification is used in a program where a value of some specific code specification type is required, its context specification and target type can be omitted. For example, in the program below, the type of the statement specification in line 1 should match to the type of the left-hand side of the assignment, which is `code_spec<int x; return int>`. Similarly, the type of the expression specification in line 2 is deduced to `double exp_spec<int x>`.

```
code_spec<int x; return int> c1 = ‘{ return x; };
double exp_spec<int x> c2 = ‘(x + 1.5);
```

The places where (and how) the type of the specifications can be deduced are following:

1. Right-hand side of assignments (type deduced from left-hand side)
2. Initializers in variable declarations (from the declared type)
3. An argument of `return` (from the return type of enclosing method)
4. Inside `? :` expressions, where above rules applies for the type of `? :` expressions (from the deduced type of the expressions)

2.3 Embedding another code specification

In the body of a code specification, another code specification can be embedded by writing `@` followed by an identifier. The code generated from inner code specification is inlined into the code from outer code specification. For example, in the code below, compiling `c2` will generate the almost the same code as one generated from `c3`.

```
code_spec<String g, x> c1 =
     ‘{ System.out.println(g + x + "!"); };

code_spec<String g; return void> c2 =
     ‘{ String x = "Michael"; @c1; return; };


code_spec<String g; return void> c3 =
     ‘{
        String x = "Michael";
        System.out.println(g + x + "!");
        return;
      };
```

When a code specification is embedded by `@`, the context specification of inner specification is always checked against surrounding code and the context of outer specification, to ensure that the composed code is type safe. In the above example, `c1` requires that variables `g` and `x` must be bound to type `String`. The code surrounding `@c1` in `c2` provides binding of `x`. `g` is not bound by `c2` itself, but it requires outer context to bind `g`. Therefore, type-checking above code succeeds.

In addition to variables, labels (or break points) can also be “free”. In DynJava, anonymous break point, which is provided by loop constructs without label, is treated as “null label”. `Break` and `continue` statements may point to labels which are bound outside current code specifications. In the following program, `break` in `c1` makes the program escape from the `for` loop in `c2`.

```
code_spec<break; int x> c1 =
     ‘{ if (x == 5) break; }

code_spec<> c2 =
     ‘{ for(int x = 0; x < 10; x++) {
           System.out.print(" " + x);
           @c1;
        }
      };
```

2.4 Embedding constant primitive values

Primitive values can also be embedded (or “lift”ed), by using `$`-prefix.

```
String message = "hello";
code_spec<String x> c1 =
     ‘{ System.out.println($message + ", " + x + "!"); };
code_spec<return void> c2 =
     ‘{ String x = "Michael"; @c1; return; };
      // print "hello, Michael!" and escape from current method
```

The expression `$message` in above program embeds runtime value of the variable `message` in to the code specification `c1`. The values which can be embedded by the `$`-expression are limited to primitive values and strings. This reflects a limitation of Java language and Java virtual machine.

2.5 Class specifications

In DynJava, code specifications must be compiled into class to use. In order to generate a class, DynJava provides class specification constructs which begins with keyword `class_spec`. A class specification looks like a class definition that lacks bodies of methods, but its instance acts as a “generator of a new class”. The code below is a small example of a class specification.

```
 1  // "interface"
 2  abstract class Method { abstract void invoke(); }
 3
 4  class_spec MethodGen extends Method {
 5    void invoke(); // this class overrides invoke() in class Method
 6  }
 7
 8  public class Test {
 9    public static void main(String[] args) {
10      MethodGen cs = new MethodGen();
11      cs.<void invoke()> = ‘{ System.out.println("Hello"); };
12
13      cs.compile(); // generates class
14
15      Method m = new cs(); // generates instance
16      m.invoke();
17    }
18  }
```

In the class specifications there is a field for each declared methods.

To define the actual body of the methods, the user assigns code specifications to the fields of class generator i.e. an instance of a class defined by `class_spec`). The fields of the class generators are indicated with an extended syntax, `e.`<*method signature*>, which appears in line 11 above. These fields have appropriate types assigned by the type checker. In the example, `cs.<void invoke()>` has the type `code_spec<extends Method; return void; void invoke()>`.

To generate an instance of dynamically-generated class, extended form of `new` expression, which takes a class generator rather than class name as an argument is used (see line 15). It returns a reference to new instance, which is typed to the base type declared in the class specification declaration. If the code specification is not compiled explicitly, it is automatically compiled at the first call to `new`.

In addition to above syntax, DynJava allows to create an anonymous class generator inside a method. If the keyword `class_spec` is used without class name, and with the variable declaration after declaration body, It generates an instance of an anonymous class specification directly. Above example can be rewritten using anonymous class specification as follows:

```
// "interface"
abstract class Method { abstract void invoke(); }

public class Test {
  public static void main(String[] args) {
    class_spec extends Method { void invoke(); } cs;

    cs.<void invoke()> = ‘{ System.out.println("Hello"); };

    // cs.compile(); // can be omitted
    Method m = new cs(); // generates class and instance
    m.invoke();
  }
}
```

2.6 Syntactic sugar

Because a notation of context specification is generally very long, and same context specification appears many time in program, DynJava provides a syntactic sugar. If a context specification contains a context of another code specification as a subset, the common elements can be abbreviated by a term like `@e`. For example, in the program shown in the previous section, the type notation `code_spec<@(cs.<void invoke();>)>` represents the type of the code specification in the example. The abbreviation can be used with other elements, like < `@e; int x, y;` >.

3 Type System

DynJava statically type-checks each code specification using its context information so that the dynamically composed codes preserves type safety.

3.1 Sub-context relation

Firstly, Sub-context relation between two contexts is defined. $\Gamma' \prec_C \Gamma$ reads as "the context Γ' is sub-context of Γ" and means that any code specification depending on Γ' can be embedded in positions that supplies Γ. The relation is defined as a context pairs that satisfies all of the following conditions:

1. Base class of Γ' is a *superclass* of base class of Γ.
2. All names provided in Γ' are also defined and has the same type in Γ.
3. All methods and constructors in Γ' are also defined and have the same signatures. Also, a superclass of each elements in the set of exceptions thrown by the method in Γ' appears in the set of exceptions thrown by the corresponding method in Γ.
4. If Γ' has a return type, Γ also has the same return type.
5. Exceptions declared to be thrown by Γ' must be subset of those of Γ, considering subclass relations.
6. All labels in Γ' are defined in Γ. In this rule, request of `break` in Γ' may be satisfied by `continue` in Γ, and unlabeled `break` and `continue` may be satisfied by labeled one.
7. If and only if Γ is in constructor state, Γ' must be in constructor state.
8. If Γ' is switch state, Γ must be in switch state.

$$\frac{R; \Gamma \vdash l : \textit{statement list}}{\Gamma; \circ \vdash \texttt{`<}R\texttt{>\{}l\texttt{\}} : \texttt{code\_spec<}R\texttt{>}} \quad \text{(TR-CSpec)} \qquad \frac{R; \Gamma \vdash e : t}{\Gamma; \circ \vdash \texttt{`}t\texttt{<}R\texttt{>(}e\texttt{)} : t\ \texttt{exp\_spec<}R\texttt{>}} \quad \text{(TR-ESpec)}$$

$$\frac{\Delta(x) = \texttt{code\_spec<}R\texttt{>} \qquad R \prec_C \Gamma}{\Gamma; \Delta \vdash \texttt{@}x\texttt{;} : \textit{statement}} \quad \text{(TR-EmbedS)} \qquad \frac{\Delta(x) = t\ \texttt{exp\_spec<}R\texttt{>} \qquad R \prec_C \Gamma}{\Gamma; \Delta \vdash \texttt{@}x : t} \quad \text{(TR-EmbedE)}$$

$$\frac{\Gamma(x) = t}{\Gamma; \Delta \vdash x : t} \quad \text{(TR-VarRef)} \qquad \frac{\Delta(x) = t}{\Gamma; \Delta \vdash \texttt{\$}x : t} \quad \text{(TR-Lift)}$$

$$\frac{\Gamma; \Delta \vdash x : \texttt{code\_spec<}R\texttt{>} \quad R \prec_C R'}{\Gamma; \Delta \vdash x : \texttt{code\_spec<}R'\texttt{>}} \quad \text{(TR-Coerce-CS)} \qquad \frac{\Gamma; \Delta \vdash x : t\ \texttt{exp\_spec<}R\texttt{>} \quad R \prec_C R'}{\Gamma; \Delta \vdash x : t\ \texttt{exp\_spec<}R'\texttt{>}} \quad \text{(TR-Coerce-ES)}$$

Figure 1: Type judgement rules for DynJava (1): dynamic codes

3.2 Type judgements

A type judgment $\Gamma; \Delta \vdash e : t$ determines that an expression or statement e has type t under current environment Γ and *outer* environment Δ.

Dynamic code specifications Firstly, we show the typing rules which relate to dynamic code generation constructs. Figure 1 shows the typing judgements for `‘`, `@` and `$` constructs. As (Tick-CSpec) and (Tick-ESpec) rules show, the context of the type of the code specification is used as a current environment for the body of the specification, which is supposed to supply all name, method, and label bindings, etc. $\Gamma(x)$ means the type of name x, which is bound syntactically most recently. In usual Java program, Δ is always unavailable, notated as $\circ$. Δ is bound to other context by (Tick-CSpec) and (Tick-ESpec), and used by (Tick-EmbedS), (Tick-EmbedE) and (Tick-Lift).

In rules (TR-Cspec) and (TR-Espec), current context for l and e is bound from the context specification given in program, and old current context is bound to outer context. Outer context Δ is referred by rules (TR-Lift), (TR-EmbedE), (TR-EmbedS) which correspond to `$` and `@`, while normal variable reference expression refers to Γ by (VarRef). Whenever a code specification is embedded by `@`, the context of inner specification is always compared with the current context provided by outer specification by $\succ_C$, to ensure that the composition is correct.

For each field of the `class_spec`, an appropriate type is automatically assigned by the type checker. In the example in previous section, `cs.<void invoke()>` has given the type `code_spec<extends Method; return void; void invoke()>`.

Statement block Nextly, we show the typing judgement rules for various usual Java constructs. These rules are approximately a re-implementation of Java's typing rules, using the concept of extended context in DynJava. As space is limited, we present only a portion of the typing rules in this paper. Detailed typing rules will be published elsewhere [10].

Rules in figure 2 are the type judgements for statement block and method body. The first five rules implements that constructor call may appear only at the top of constructor body, and unless must be there if superclass constructor without argument is available.

Labels Rules in Figure 3 are for while, break and continue statements. These rules implements the requirement for existence of enclosing loop constructs with the label environment in Γ. (TR-While-L) adds the label binding `continue` l into Γ, and (TR-Break) and (TR-Continue) checks it. The meaning of relation $\in_L$ is that if the `continue` l is allowed at some program points, `break` l is always allowed in Java language.

$$\frac{\texttt{super} \notin \Gamma}{\Gamma; \Delta \vdash \varepsilon : \textit{statement list}} \tag{TR-StmtListNil}$$

$$\frac{\texttt{super} \notin \Gamma \qquad \Gamma; \Delta \vdash l : \textit{statement list} \qquad \Gamma; \Delta \vdash s : \textit{statement}}{\Gamma; \Delta \vdash s;\ \ l \vdash \textit{statement list}} \tag{TR-StmtList}$$

$$\frac{\texttt{super} \notin \Gamma \qquad \Gamma^*, t\ x; \Delta \vdash l : \textit{statement list}}{\Gamma; \Delta \vdash t\ x; l : \textit{statement list}} \tag{TR-VarBind}$$

$$\frac{\begin{array}{c}\texttt{super} \in \Gamma \qquad \Gamma; \Delta \vdash s : \textit{constructor call} \\ \Gamma \setminus \{\texttt{super}\}; \Delta \vdash l : \textit{statement list}\end{array}}{\Gamma; \Delta \vdash s;\ \ l : \textit{statement list}} \tag{TR-ConstrBody-1}$$

$$\frac{\begin{array}{c}\texttt{super} \in \Gamma \qquad \texttt{super()} \in \Gamma \\ \Gamma \setminus \{\texttt{super}\}; \Delta \vdash l : \textit{statement list}\end{array}}{\Gamma; \Delta \vdash l : \textit{statement list}} \tag{TR-ConstrBody-2}$$

$$\frac{\Gamma^*; \Delta \vdash l : \textit{statement list}}{\Gamma; \Delta \vdash \{l\} : \textit{statement}} \tag{TR-Block}$$

(The environment Γ^* means the Γ with switch state flag cleared.)

Figure 2: Type judgement rules for DynJava (2): statement list

3.3 Alternative sub-context relations

The relation $\prec_C$ is defined to satisfy that if both $\Gamma, \circ \vdash e : t$ and $\Gamma \prec_C \Gamma'$ are satisfied, $\Gamma', \circ \vdash e : t$ is satisfied. If the all syntactic restrictions of the target language are fully written down as a form of type judgement on context (Γ), and if the relation $\succ_C$ which satisfies the above relation, the typing system for dynamic code fragments could be derived from the typing rules of base language, as shown in this paper. However, such derived rules are sometimes not convenient for actual use.

For example, Java language requests that all variables must be properly initialized before its use, with regard to all possible syntactic execution paths. That restriction can be introduced into DynJava's typing rule. However, if it is done, the typing rules become very complicated. Especially, the user must always specify the correct state of variable initialization for every code specifications, which is hard to maintain. We designed DynJava to avoid that difficulties by assuming and ensuring that the variables declared in the code specifications are always initialized by default value (0 or null).

4 Implementation

In this section, we present our current implementation of the DynJava language. Our compiler system consists of two parts: the language preprocessor and the code postprocessor, which utilizes Java's original compiler `javac` as a back-end code generator. This approach is similar to those of Tempo [2, 11].

Firstly, the language preprocessor reads the source code and type-checks the whole code. For each code specification and expression specification, the preprocessor generates a code generator class, with a *template* of dynamic code in the Java language. The template is then processed by the `javac` and compiled into byte code. The code postprocessor reads the result of `javac` and translates it into Java program which generates the bytecode on runtime compilation time.

The template contains the code very similar to the original code appeared in the specifications. For example, the template generated from simple expression specification `'int<>(5 + 3)` is like following:

```
    private void __template() { return (5 + 3); }
```

It is compiled into the bytecode which pushes the result of 5 + 3 onto the stack, which can be used "verbatimly" in the dynamically-generated code. However, `@`- and `$`-expressions are not embeddable

$$\frac{\Gamma; \Delta \vdash e : \texttt{boolean} \qquad \Gamma^*, \texttt{continue}\ l; \Delta \vdash s : \mathit{statement}}{\Gamma; \Delta \vdash l\texttt{:}\ \texttt{while}\ \texttt{(}e\texttt{)}\ \ s : \mathit{statement}} \quad \text{(TR-While-L)}$$

$$\frac{\texttt{break} \in_L \Gamma}{\Gamma; \Delta \vdash \texttt{break;} : \mathit{statement}} \quad \frac{\texttt{break}\ \ l \in_L \Gamma}{\Gamma; \Delta \vdash \texttt{break}\ \ l\texttt{;} : \mathit{statement}} \quad \text{(TR-Break)}$$

$$\frac{\texttt{continue} \in_L \Gamma}{\Gamma; \Delta \vdash \texttt{continue;} : \mathit{statement}} \quad \frac{\texttt{continue}\ \ l \in_L \Gamma}{\Gamma; \Delta \vdash \texttt{continue}\ \ l\texttt{;} : \mathit{statement}} \quad \text{(TR-Continue)}$$

$$\frac{r \in L}{r \in_L L}$$

$$\frac{\texttt{continue}\ \ l \in_L L}{\texttt{break}\ \ l \in_L L}$$

$$\frac{\texttt{break}\ \ l \in_L L}{\texttt{break} \in_L L} \quad \frac{\texttt{continue}\ \ l \in_L L}{\texttt{continue} \in_L L}$$

Figure 3: Type judgement rules for DynJava (3): loop-related constructs

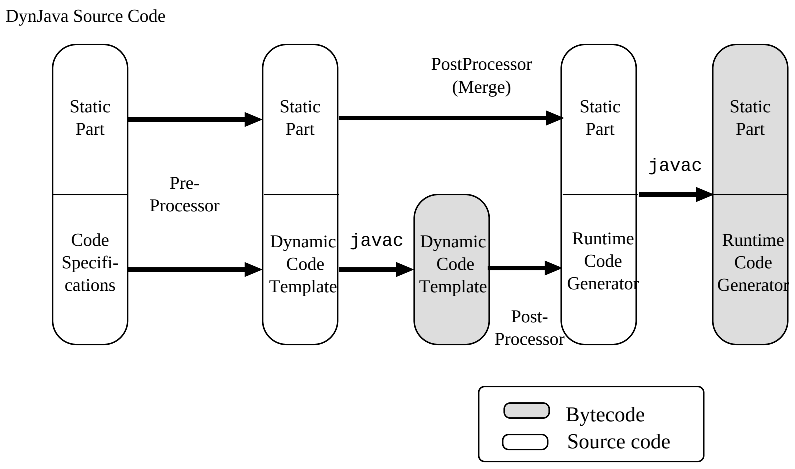
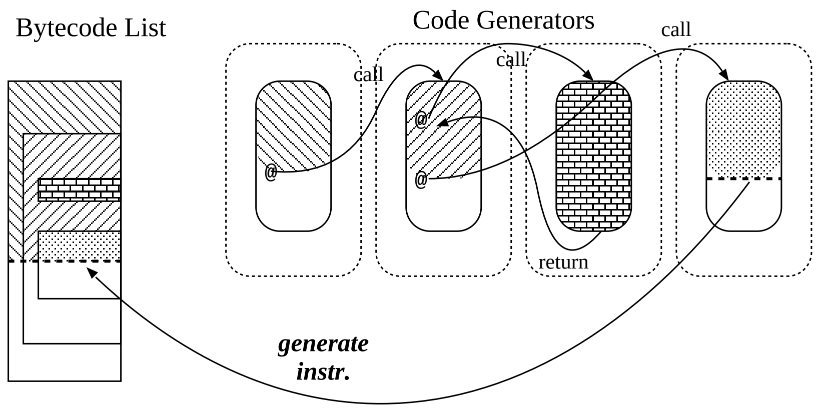


Figure 4: Process flow of DynJava compiler

into the template directly, and many other constructs like method invocation of current class, non-local `break`/`continue`, etc. are the same. For such constructs, preprocessor generates some dummy constructs into the implementation class, and translate those constructs into some javac-compatible code. The code postprocessor knows about specific bytecode patterns generated from such “fake” constructs, and converts them appropriately.

For example, for each `@`-expression a dummy method named `__t`*n* is generated, and the expression is translated into the invocation of the dummy method. When the postprocessor finds the invocation of the method named `__t`*n*, the postprocessor translates it into the nested invocation of runtime code generator (Figure 5), rather than the invocation instruction itself.

When the user requests to generate a class and an instance from code specifications, the runtime system invokes the code generator to produce bytecode, then asks Java’s classloader to load the generated class. As Java’s bytecode architecture is stack-based, embedding some bytecode fragment into another will implement semantics of `@`-expression almost automatically. However, as each template of code uses the slots of local variables in its own way, naive merger of two codes will conflict with each other on the usage of local variable. Our postprocessor and generated code generator counts the number of used slots at each



Patterned bytecode fragments are generated by the code generator with corresponding patterns. The code generated by inner code generator is embedded into code generated by outer one.

Figure 5: Nested code generator invocation by `@`-expression.

Table 1: Execution time of one FFT calculations and its code generation time

n	960	1024	2048	3600	6561	8192	10000	16384	30030	44100	65535	1048575
#factors	8	10	11	8	8	12	8	13	6	8	3	6
T_{Dynamic}	**2.11**	**2.16**	**4.23**	**7.85**	15.08	**24.95**	25.7	**53.40**	97.0	**130.1**	899.2	6051
T_{JIT}	*2650*	*4153*	*5270*	*2296*	*2275*	*7909*	*2270*	*9366*	*1261*	*2298*	*694*	*1179*
T_{Static}	2.15	2.39	5.19	8.34	**13.90**	25.64	**25.2**	56.96	**82.3**	136.5	**878.0**	**5915**

[unit: milliseconds]

embed point and shifts the slot numbers used by inner code fragment to unused ones.

Currently, preprocessor is implemented as a plug-in for EPP [8], an extensible Java preprocessor package. Postprocessor and runtime code generator is implemented using JavaClass [4], a library for bytecode manipulation. As runtime code generator should run very fast to minimize code generation cost, we are planning to re-implement the code generator using more light-weight methods.

5 Evaluation

To evaluate the performance of DynJava implemented dynamic code, we implemented a runtime optimizer for fast Fourier transform (FFT). FFT calculates the discrete Fourier transform (DFT) of size n in the computational order less than $O(n^2)$. There are many known algorithm to optimize the calculation of DFT [6], but all of them are depending on the size n.

Our prototype optimizer implements the Cooley-Tukey fast Fourier transform [3], along with a naive DFT routine. The Cooley-Tukey algorithm is applicable when the size n is factored into $n_1 n_2$, and reduce the DFT calculation to small DFT calculations of size n_1 and n_2. If n is factored into more than 2 prime numbers, the algorithm can be applied recursively. Our optimizer factors n into primes and generates a specialized method in which the all nested DFT calculations are inlined. We also implemented a routine which implements the same algorithm using an object for representation of nested DFT calculations and compared the performance.

Table 1 shows the calculation time for DFT of various sizes. Experiments are done on the IBM build of JDK 1.3.0, running PentiumIII at 500MHz. The data shown as T_{Dynamic} are the execution time of dynamically-generated code, and T_{Static} are those of the routine using object representation. T_{JIT} is the times expensed for JIT compilation, which are guessed by the difference between the times consumed by the first invocation and other invocations. When n has many small factors, especially 2, the dynamically-generated code runs significantly faster than static one. The times consumed by JIT compilation roughly depends on the number of factors, and it is around 2.3 seconds for various n with 8 prime factors, and 9.3 seconds when $n = 16384 = 2^{14}$.

6 Related Work

There are many tools which manipulates bytecode, for example JavaClass API [4], and `gnu.bytecode` package included in Kawa Scheme [1], and they can be directly used to generate execution code dynamically on Java Virtual Machine (JVM). However, as those tools treat a dynamically generated program as a stream of untyped instructions, the user could generate type-unsafe code. Keeping safety of the generated code completely owe to the user's responsibility, and generally it is very hard to maintain.

'C [11] is an extension to C language which supports writing dynamic code in the syntax of C language. In 'C, user writes fragments of dynamic code and combine them to generate a function in C language at runtime. However, as 'C does not support any context, type safety of the generated code is still not guaranteed enough. For example, two code fragments `'{return 5;}` and `'{return "string";}` cannot co-exist in one function, but as these two specifications are both typed "`void cspec`" in 'C, the inconsistency is not detected. Our DynJava can detects those inconsistency at compilation time using context information. In addition to this, in 'C users must write a special construct explicitly to use variables or labels across two or more code fragments, and must maintain the consistency of them carefully. DynJava provides easier way to share one variable between two specifications than one provided by 'C.

Another approach to generate code dynamically in a type-safe way is a runtime specialization technique, that uses program analysis. For example, the second and third authors describes about runtime program specialization on Java bytecode [9], and many related publications on this topic exist. Since this approach extracts dynamic code fragments from a given single-level program, specialized program is always type-safe. However, degree of optimizations (specialization) depends on the preciseness of the analysis. If the target program is simple, full-automatic analysis is sufficient produce a good result. However, if the program gets complicated, the program author will have better knowledge about program's property and where to optimize than the automatic analyzer. In this case, our DynJava will become a powerful tool to implement program-dependent runtime optimization easily by hand.

There are studies on of type safety of dynamic program composition. Modal-ML [12] is one of such studies on the functional language ML. Due to simple syntax and semantics, restriction on a correct context for a dynamic code fragments can be easily checked by matching types of all free variables. However, in imperative languages such as Java, context should have more precise information to determine the correctness, because a program fragment depends not only on the types of free variables, but also various information such as labels and exceptions. Our type system is extended to handle these properties and ensures the correctness of composition in an imperative language.

MobileML [7] defines type system for dynamically-bound code fragments on the ML language, using a notion of context. The base idea of context type checking in DynJava is inspired by Mobile-ML. However, its context notion binds only variables, because of the same reason as Modal-ML.

7 Conclusion

We presented a strongly typed language that supports dynamic code generation. The user can write dynamic code fragments using high-level language constructs. By introducing a context which holds various syntactic information as well as variable bindings, DynJava's type system statically guarantees the type-safety of that dynamically composed code fragments. Our current implementation demonstrates that the system can be used to easily implement dynamic optimization of a FFT program.

Acknowledgments

We are thankful to Dr. Naoki Kobayashi and Dr. Kenjiro Taura for many valuable suggestions on this research.

References

[1] Per Bothner. Kawa—compiling dynamic languages to the Java VM. In *USENIX*, New Orleans, June 1998.

[2] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[4] Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Institut für Infomatik, Freie Universität Berlin, 7 July 1998.

[5] Dawson R. Engler. VCODE: A retargetable, extensive, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 160–170, Philadelphia, PA, USA, May 1996.

[6] Matteo Frigo. A fast Fourier tranform compiler. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 169–180, Atlanta, GA USA, May 1999.

[7] Masatomo Hashimoto and Akinori Yonezawa. MobileML: A programming language for mobile computation. In *Proceedings of the 4th International Conference on Coordination Languages and Models (COORDINATION 2000)*, number 1906 in Lecture Notes in Computer Science, pages 198–215. Springer-Verlag, 2000.

[8] Yuuji Ichisugi. Epp homepage. `http://www.etl.go.jp/~epp/`.

[9] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization: A portable approach to generating optimized specialized code. In Olivier Danvy and Andrzej Filinski, editors, *Second Symposium on Programs as Data Objects (PADO II)*, In Lecture Notes in Computer Science. Springer-Verlag, Aarhus, Denmark, May 2001. To appear.

[10] Yutaka Oiwa. A Java-based language with type-safe dynamic-code generation. Master's thesis, Graduate School of Science, the University of Tokyo, February 2001. to appear.

[11] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):324–367, March 1999.

[12] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and modal-ML. In *the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 224–235, 1998.